AFRL-RI-RS-TR-2017-077

A PLUG-AND-PLAY ARCHITECTURE FOR PROBABILISTIC PROGRAMMING

LOGICBLOX, INC.

APRIL 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-077 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
JOHN D. MATYJAS
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
APRIL 2017	FINAL TECHNICAL REPORT	JUN 2014 – JUN 2016

4. TITLE AND SUBTITLE

A PLUG-AND-PLAY ARCHITECTURE FOR PROBABILISTIC PROGRAMMING

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-14-2-0217

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Molham Aref, Yannis Kassios, Benny Kimelfeld, Emir Pasalic, and Zografoula Vagena

5d. PROJECT NUMBER
PPML

5e. TASK NUMBER
BL

5f. WORK UNIT NUMBER
OX

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
LogicBlox, Inc
1349 West Peachtree St NW
Atlanta, GA 30309

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2017-077

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2017-1437
Date Cleared: 31 Mar 2017

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In the probabilistic-programming paradigm, the application logic is specified by means of a description of a probabilistic model (by stating how a sample is being produced) using a Probabilistic Programming Language (PPL). The principal value one obtains from a probabilistic program lies in the inference thereof, that is, reasoning about the entire probability distribution that the program defines (e.g., finding a likely event or estimating its marginal probability). The PPAML kickoff meeting highlighted several research challenges regarding the development of inference infrastructure for PPL, for both increasing software efficiency and reducing software complexity, towards the goal of broadening the PPL applications and the community of implementers and programmers. These challenges include the design of an Application Program Interface (API), or alternatively an Intermediate Representation Language (IRL), that would allow new solvers to be plugged into existing PPLs, and for PPL engines to be able to pick from and combine solvers for a given problem.

15. SUBJECT TERMS

Probabilistic programming, statistical solvers, Datalog

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILMAR SIFRE**
U	U	U	UU	42	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures and Tables

Summary

In the probabilistic-programming paradigm, the application logic is specified by means of a description of a probabilistic model (by stating how a sample is being produced) using a Probabilistic Programming Language (PPL). The principal value that one obtains from a probabilistic program lies in the inference thereof, that is, reasoning about the entire probability distribution that the program defines (e.g., finding a likely event or estimating its marginal probability). The PPAML kickoff meeting highlighted several research challenges regarding the development of inference infrastructure for PPLs, for both increasing software efficiency and reducing software complexity, towards the goal of broadening the PPL applications and the community of implementers and programmers. These challenges include the design of an Application program interface (API), or alternatively an Intermediate Representation Language (IRL), that would allow new solvers to be plugged into existing PPLs, and for PPL engines to be able to pick from and combine solvers for the problems induced by the given program.

The work of LogicBlox within the DARPA agreement consisted of two phases. In the first phase, we have designed and implemented tools for mapping PPLs (TA2) into solvers (TA3). Specifically, we have worked with the PPLs BLOG and Chimple, and the solvers PRAiSE and BBVI (Black-Bock Variational Inference). The challenges we have faced in designing the translations have led us to seek an appropriate IRL. We focused our search on logic-based, declarative languages, that we can approach with our expertise derived from developing the LogicBlox product. This thought process has led us to the conclusion that our product would greatly benefit from a probabilistic programming capability. To be integrated with our product, such capability should extend Datalog – the database query language that our language, LogiQL, is based upon. Nevertheless, it turns out that using existing approaches that incorporate randomness into logic force us to compromise some inherent principles of Datalog. Consequently, in the second phase of the DARPA agreement we have focused on developing a natural probabilistic-programming extension of Datalog, namely Probabilistic Programming Datalog (PPDL), and its implementation as an extension of LogicBlox's query language LogiQL, namely PlogiQL.

1 Introduction

In the probabilistic-programming paradigm, the application logic is specified by means of a description of a probabilistic model (by stating how a sample is being produced) using a Probabilistic Programming Language (PPL). Machine-learning solutions are often described in this way: a probabilistic story describes how data is generated (the *model*), and the training examples are treated as evidence to infer the randomly selected parameters (by *conditioning*). The principal value one obtains from a probabilistic program lies in the inference thereof, that is, reasoning about the entire probability distribution that the program defines (e.g., finding a likely event or estimating its marginal probability). The PPAML kickoff meeting highlighted several research challenges regarding the development of inference infrastructure for PPL, for both increasing software efficiency and reducing software complexity, towards the goal of broadening the PPL applications

and the community of implementers and programmers. These challenges include the design of an API, or alternatively an Intermediate Representation Language (IRL), that would allow new solvers to be plugged into existing PPLs, and for PPL engines to be able to pick from and combine solvers for a given problem.

Our work in the DARPA PPAML program consisted of two phases. In the first phase we designed and implemented tools for mapping PPLs (the focus of PPAML Technical Areas 2) into solvers (the focus of Technical Area 3). Specifically, we have worked with the PPLs BLOG [9] and Chimple [8], and the solvers PRAiSE [12] and BBVI [10] (Black-Bock Variational Inference) . PRAiSE is a system that supports a logical language for defining factor graphs via first-order logic, just like Markov Logic Networks (MLNs). BBVI realizes the basic idea of *variational inference*: replace the posterior (original) probability distribution with an alternative (variational) probability distribution of a simpler family, such that the variational distribution is "closest" to the original distribution. The translations currently restrict the languages, mainly due to missing components of PRAiSE and challenges in the utilization of the BBVI software.

The challenges we have faced in designing the translations have led us to seek an appropriate IRL. We focused our search on logic-based, declarative languages, that we can approach with the expertise and machinery derived from developing the LogicBlox product [1]. To be integrated with our product, the probabilistic-programming capability should extend Datalog – the database query language that our language, LogiQL [6], is based upon. To enhance Datalog with the capability of probabilistic programming, while maintaining the inherent features of Datalog, an extension should satisfy three elementary conditions.

1. **Compositional probabilistic programming**. Given the importance of parameter selections in motivation of probabilistic programming, the extension should allow for random number generation. More importantly, we should be able to use the random numbers (constructed via the logic of the program) as the parameters of subsequent numerical distributions.
2. **Recursion through randomness**. The expressiveness of Datalog is based on its support for recursion (fixpoint semantics). While it is easy to define the semantics of probabilistic Datalog programs without recursion, or with stratification (as often done with extensions of Datalog and Prolog), it is quite nontrivial to define the semantics of programs that support both random number generation and unlimited recursion.
3. **Invariance under logical equivalence**. Datalog engines use the logical nature of the program in order to do program rewriting for performance optimization, where correctness is guaranteed as long as logical equivalence is preserved. A logical program with randomness can be thought of as a First Order Logic (FO) program with function calls, and as such, we would like it to retain invariance under equivalence.

Interestingly, it turns out that using existing approaches that incorporate randomness into logic violate at least one of the above three conditions. For example, languages that follow the "distributional semantics" such as ProbLog [7] and PRISM [11] violate the first

condition, logic based Probabilistic Programming Systems (PPSs) such as BLOG [9] violate the second condition, and template-grounding languages such as MLN [5] and PRAiSE [12] violate the third.[1]

Consequently, in the second phase of the DARPA agreement we have focused on developing a natural probabilistic-programming extension of Datalog, namely Probabilistic Programming Datalog (PPDL), and its implementation as an extension of LogicBlox's query language LogiQL, namely PlogiQL. The research of PPDL consisted of two main efforts. The first was the design and development of the theoretical foundations. We have published a paper [1] that describes our proposal in the International Conference on Database Theory (ICDT),[2] which is one of the two most important international conferences on database theory. The second was the design and development of an extension of the LogicBlox query language, LogiQL, with the theoretical PPDL concept, namely the PLogiQL language.

2 Methods, Assumptions, and Procedures

We now describe our effort in the PPAML program in detail. We divide the discussion into the first phase and the second phase.

2.1 First Phrase: Connecting Languages to Solvers

In the first phase, we have designed and implemented tools for mapping PPAML PPLs into PPAML solvers. We have worked with the PPLs BLOG and Chimple, and the solvers PRAiSE and BBVI (Black-Bock Variational Inference).

2.1.1 Translating BLOG to PRAiSE

PRAiSE is a system that supports a logical language for defining factor graphs via first-order logic, just like MLNs. A technical difference between PRAiSE and MLNs is in the way the factors are defined: in PRAiSE every grounded formula gives rise to a direct factor, which is the weight of the formula; in MLNs, this factor is exponential in the weight. In particular, defining a hard constraint in PRAiSE amounts to assigning the weight 0 to the formula that defines the negation of the constraint. In terms of inference, PRAiSE is using *lifted inference*, and reverts to *loopy belief propagation* (deterministic heuristic with a possible error) when the former fails to solve the inference problem. Following is an example of a PRAiSE program, based on an example from the BLOG Language Reference.

```
/* Define a class of type House, having precisely 4 instances. Among the instances
are the ones mentioned here by name (we could name only some of the houses).
*/
sort House: 4, maryhouse, johnhouse, cathyhouse, rogerhouse;

/* Define a random variable earthquake. */
```

[1] Here we are using the classification of probabilistic logic by De Raedt and Kimmig [7].
[2] The paper has been invited by the ACM Journal on Transactions on Database Systems as one of the best papers in ICDT 2016, and a journal article is currently under submission.

```
random earthquake: -> Boolean;

/* Define a random variable burglary and a random variable alarm associated to
every House.  Hence, we effectively define 8 random variables, where every House
has a burglary and alarm variables as attributes. */
random burglary: House-> Boolean;
random alarm: House-> Boolean;

/* Add a factor 0.002 to earthquake=true and a factor 0.003 to every
burglary=true. */
earthquake 0.002;
burglary(H) 0.003;

/* Define factors for different combinations of burglary(H), alarm(H) and
earthquake. Note that we get 4 sets of factors --- one for each house. */
if burglary(H) then if earthquake then alarm(H)  0.9 else alarm(H)  0.8 else if
earthquake  then alarm(H)  0.4 else alarm(H)  0.01;

/* Define factors to different cases of alarm. Since these factors are 1 and 0, they
effectively define observations. */
alarm(maryhouse) 1;
alarm(johnhouse) 1;
alarm(cathyhouse) 1;
alarm(rogerhouse) 0;

/* The query gives the marginal probability of earthquake=true. The probability
space is defined by normalizing across all possible worlds, where the non-
normalized probability of each possible world is the product of the factors that fire
in the world.*/
Query:  earthquake
```

A BLOG program is a declaration of random variables, each identified by a type, name, distribution family, and distribution parameter. Following is an example.

```
/* Define a class of type House. */
type House;

/* Define 4 named instances of House. */
distinct House Maryhouse, Johnhouse, Cathyhouse, Rogerhouse;

/* Define a random variable “earthquake” */
random Boolean earthquake ~ BooleanDistrib(0.002);

/* Define a random variable “burglary” for each house. Here, burglary
is  notationally a function of a House instance. */
random Boolean burglary(House h) ~ BooleanDistrib(0.003);
```

```
/* Define an "alarm" random variable for each House. The actual distribution of
alarm(h) is determined by burglary(h) and earthquake, using the "case" statement.
*/
random Boolean alarm(House h) ~
      case [burglary(h), earthquake] in {
            [false, false] -> BooleanDistrib(0.01),
            [false, true]  -> BooleanDistrib(0.40),
            [true, false]  -> BooleanDistrib(0.80),
            [true, true ]  -> BooleanDistrib(0.90)
       };

/* Observations (defining the posterior probability). */
obs Alarm(Maryhouse) = true;
obs Alarm(Johnhouse) = true;
obs Alarm(Cathyhouse) = true;
obs Alarm(Rogerhouse) = false;

/* Query: find the marginal probability of Earthquake=true. */
query Earthquake;
```

The translation from BLOG to PRAiSE is done line by line. Types are translated to Sorts, and Boolean variables are translated into random variables and corresponding factors. There are two intricacies in the current state of the translation.

Translating distributions in conditions. When translating a distribution within a conditional statement such as "if" and "case," each distribution becomes a factor, with its enclosing conditions being translated into Boolean conditions. Note that there can be any number of levels of condition nesting in BLOG. An example follows.

BLOG	PRAiSE

random Boolean U ~ BooleanDistrib(0.3);
random Boolean V ~ BooleanDistrib(0.9);
random Boolean W ~ BooleanDistrib(0.1);
	random U: -> Boolean;
	U 0.3;
	random V: -> Boolean;
	V 0.9;
	random W: -> Boolean;
	W 0.1;

random Boolean X ~
  if U then
  case W in {
  true -> BooleanDistrib(0.8),
  false -> BooleanDistrib(0.2)
  }
  else
  case V in {
  true -> BooleanDistrib(0.8),
  false -> BooleanDistrib(0.2)
  };
	random X: -> Boolean;
	
	if U then
	  if W then X 0.8
	  else X 0.2
	else
	  if V then X 0.8
	  else X 0.2;

Translating categorical distributions. Since PRAiSE supports only Boolean (random) variables, it does not have any type that corresponds directly to a *categorical* random variable, that is, a variable that is defined by an explicitly stated distribution over a finite set of values. An example follows.

BLOG	PRAiSE
type Student; distinct Student John, Mary, Fred;	sort Student: 3, John, Mary, Fred;
type Level; distinct Level Weak, Average, Smart;	sort Level: 3, Smart, Average, Weak;
random Level intelligence(Student s) ~ Categorical({ Smart -> 0.3, Average -> 0.5, Weak -> 0.2});	random intelligence_smart: Student-> Boolean; random intelligence_average: Student-> Boolean; random intelligence_weak: Student-> Boolean; */* Define a dummy variable to get the factors right. */* random dummy: -> Boolean; dummy 1; */* Factor 0.3 for (1,0,0)*/* if intelligence_Smart(S) and not intelligence_Average(S) and not intelligence_Weak(S) then dummy 0.3 */* Factor 0.5 for (0,1,0)*/* else if not intelligence_Smart(S) and intelligence_Average(S) and not intelligence_Weak(S) then dummy 0.5 */* Factor 0.2 for (0,0,1)*/* else if not intelligence_Smart(S) and not intelligence_Average(S) and intelligence_Weak(S) then dummy 0.2 */* Nothing else is permitted. */* else dummy 0;
obs intelligence(Mary) = Average;	intelligence_Average(Mary);

Our implementation has been done over the source code of BLOG. In particular, we have revised the compiler to create PRAiSE code rather than BLOG's internal representation of the program.

2.1.2 Translating BLOG to BBVL

Variational inference has become a widely deployed method to approximate posterior distributions in complex latent-variable models. The underlying idea is to replace the posterior (original) probability distribution with an alternative (variational) probability distribution of a simpler family, such that the variational distribution is "closest" to the original distribution, in the following sense.

In a probabilistic model P(x|z), let x be the vector of observations, z be the vector of latent variables, and λ the vector of free parameters of a variational distribution Q(z | λ). The goal is to approximate P(x|z) with a setting of λ. This is achieved by minimizing the KL divergence:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \cdot \ln \frac{P(i)}{Q(i)} \quad (1)$$

Or, equivalently, maximizing the Evidence Lower BOund (ELBO), which is proportional to the reciprocal of the KL divergence.

Deriving a variational-inference algorithm generally requires significant and strenuous model-specific analysis and computation, which hinders its widespread acceptance and usage. Black-Box Variational Inference (BBVI) [10] employs stochastic optimization to maximize ELBO. The basic BBVI algorithm requires the ability to:

1. Compute the logarithm of the joint distribution P(x, z) at a given input point.
2. Sample from the variational distribution Q(z | λ).
3. Compute the logarithm of the variational distribution Q(z | λ) at a given input point.
4. Compute the gradient of the logarithm, as a function of λ, of the variational distribution Q(z | λ) at a given input point.

In particular, in BBVI the model-specific computations (i.e., those that depend on the distribution P) are restricted to the computation of point probabilities in the prior distribution (which is typically tractable).

Due to lack of a publicly available implementation of the BBVI method, as part of our work we have implemented (in Java) the basic BBVI algorithm (i.e. without the variance-reduction optimizations) and investigated its behavior on a variety of inputs. Our current implementation supports a number of uni-dimensional (i.e. Bernoulli, Beta, Gamma, Inverse Gamma, Normal, Poisson), multi-dimensional (i.e. Dirichlet, Multinomial, Multivariate Normal) as well as arbitrary factorizations over those. We have tested our implementation on two well-known latent-variable models (namely the Mixture of Gaussians and Latent Dirichlet Allocation) using a variety of generated datasets. Our preliminary observation is that the basic algorithm is highly sensitive to the initialization of the free variables (λ) and, thus, those need to be carefully chosen. We plan to further investigate and establish a collaboration with David Blei's group (Columbia and Princeton universities) around the implementation of BBVI.

Translation. Recall that in BBVI, we approximate an original distribution P(z|x) by a variational distribution q(z|λ). A BLOG program describes a probability distribution P(z|x) for some observed (assignment to) x. This is done by specifying *(a)* the joint probability distribution P(z,x), *and (b)* the observations (values assigned to x). To translate a BLOG program into BBVI, we need to (automatically):

1. Analyze the BLOG program to identify the distribution P(x, z);
2. Generate the parameterized distribution Q(z | λ);
3. Apply BBVI basic algorithm to find values for λ.

In our initial implementation, we first describe P(z,x) in a factorized form (i.e., the probability of an assignment is a product of simple functions over the assignment). We define Q(z|λ) using factors similar to P(z,x), except that the parameters of the factors are not fixed, but rather become free parameters in λ. Currently, we restrict our support to BLOG programs where none of these factors contain variables from both x and z.

We now give more details on our translation, by walking through an example. In this example, we consider the following BLOG program, which extends upon the previous BLOG example by giving assigning different behaviors to different people.

```
/*  Pearl's burglary-earthquake network, as presented in "Artificial Intelligence: A
Modern Approach", 2nd ed., p. 494.  */
random Boolean Earthquake ~ BooleanDistrib(0.002);
random Boolean Burglary ~ BooleanDistrib(0.001);
random Boolean Alarm ~
        if Burglary then
                if Earthquake then BooleanDistrib(0.95) else BooleanDistrib(0.94)
        else
                if Earthquake then BooleanDistrib(0.29) else BooleanDistrib(0.001);
random Boolean JohnCalls ~
        if Alarm then BooleanDistrib(0.9) else BooleanDistrib(0.05);
random Boolean MaryCalls ~
        if Alarm then BooleanDistrib(0.7) else BooleanDistrib(0.01);

/* Evidence for the burglary model saying that both John and Mary called.  Given
this evidence,  the posterior probability of Burglary is 0.284 (see p. 505 of "AI: A
Modern Approach", 2nd ed.) */
obs JohnCalls = true;
obs MaryCalls = true;

/* Query for the burglary model asking whether Burglary is true. */
query Burglary;
```

This program defines the following probability distribution, given in factorized form.

$$\Pr(\text{Burglary}, \text{Earthquake}, \text{Alarm}, \text{JohnCalls}, \text{MaryCalls} \mid p_1, \ldots, p_{10}) =$$

P_1(Earthquake | p_1) x
P_2(Burglary | p_2) x
P_3(Alarm | Earthquake, Burglary, p_3, p_4, p_5, p_6) x
P_4(JohnCalls | Alarm, p_7, p_8) x
P_5(MaryCalls | Alarm, p_9, p_{10})

where:

- P_1(Earthquake | p_1) = BooleanDistribution(p_1) (that is, true with probability p_1 and false with probability 1-p_1) for p_1=0.002.

- P_2(Burglary | p_2) = BooleanDistribution(p_2), p_2= 0.001

- P_3(Alarm | Earthquake, Burglary, p_3, p_4, p_5, p_6) =
  - BooleanDistribution(p_3) for p_3 = 0.95, if (Earthquake,Burglary)=(true,true)
  - BooleanDistribution(p_4) for p_4 = 0.94, if (Earthquake,Burglary)=(false,true)
  - BooleanDistribution(p_5) for p_5 = 0.29, if (Earthquake,Burglary)=(true,false)
  - BooleanDistribution(p_6) for p_6 = 0.001, if (Earthquake,Burglary)=(false,false)

- P_4(JohnCalls | Alarm, p_7, p_8) =
  - BooleanDistribution(p_7) for p_7 = 0.7, if Alarm = true
  - BooleanDistribution(p_8) for p_8 = 0.01, if Alarm = false

- P_5(MaryCalls | Alarm, p_9, p_{10}) =
  - BooleanDistribution(p_9) for p_9 = 0.9, if Alarm = true
  - BooleanDistribution(p_{10}) for p_{10} = 0.05, if Alarm = false

The program also provides evidence by fixing the values for some of the defined random variables. In the previous example, we have:

Evidence = {JohnCalls = true, MaryCalls = true} (2)

In the translation to BBVI, we first define the x and z vectors as follows. The vector x consists of all the random variables in Evidence. The vector z consists of all the random variables z_i defined in the BLOG program (in some order), such that z_i is not in Evidence. In the example program we have:

- z = (Burglary, Earthquake, Alarm)

- x = (JohnCalls, MaryCalls).

We then define the variational distribution Q(z|λ) as a multiple of factors, as follows.

For all factors p_i in P(x,z):

If p_i has the form $p_i(z)$ (i.e., no x involved), then:

- Define a factor q_i for the variational distribution Q; q_i is of the same family (but not necessarily the same parameters) as p_i
- Add the parameters of q_i in λ

In our example the variational distribution is:

$$\Pr(\text{Burglary}, \text{Earthquake}, \text{Alarm} \mid q_1, \ldots, q_6) = Q_1(\text{Earthquake} \mid q_1) \times Q_2(\text{Burglary} \mid q_2) \times Q_3(\text{Alarm} \mid \text{Earthquake}, \text{Burglary}, q_3, q_4, q_5, q_6)$$

where:

- $Q_1(\text{Earthquake} \mid q_1) = \text{BooleanDistribution}(q_1)$
- $Q_2(\text{Burglary} \mid q_2) = \text{BooleanDistribution}(q_2)$
- $Q_3(\text{Alarm} \mid \text{Earthquake}, \text{Burglary}, p_3, p_4, p_5, p_6) =$
  - $\text{BooleanDistribution}(q_3)$ if (Earthquake,Burglary)=(true,true)
  - $\text{BooleanDistribution}(q_4)$ if (Earthquake,Burglary)=(false,true)
  - $\text{BooleanDistribution}(q_5)$ if (Earthquake,Burglary)=(true,false)
  - $\text{BooleanDistribution}(q_6)$ if (Earthquake,Burglary)=(false,false)

Moreover, we have $\lambda = \{q_1, q_2, q_3, q_4, q_5, q_6\}$.

Finally, applying the BBVI algorithm with the above configuration produces values for λ. We have been incorporating the above algorithm within the BLOG codebase with the aim to reuse as much of the BLOG code as possible (e.g. BLOG scanner, BLOG defined distributions, samplers etc).

2.1.3 Translating Chimple to PRAiSE

Chimple is a Java API for defining the sampling process. The random variables are begin defined and sampled through special objects (“monkeys”) that allow the inference engine to bookkeep and manipulate the random value assignments. Following is an example.

```
public Object run(Object ... args)  {
        /* Sample Boolean (Flip) variables “burglary” and “earthquake.” */
        int burglary = chimpFlip("burglary", 0.003);
        int earthquake = chimpFlip("earthquake", 0.002);
```

```
        /* Sample alarm according to burglary and earthquake. */
        int alarm;
        if (burglary == 1) {
                if (Earthquake == 1)  alarm = chimpFlip("alarm", 0.95);
                  else alarm = chimpFlip("alarm", 0.94);
        } else {
                if (earthquake == 1) alarm = chimpFlip("alarm", 0.29);
                  else alarm = chimpFlip("alarm", 0.001);
        }
        /* Sample johnCalls according to alarm. */
        int johnCalls;
        if (alarm == 1) johnCalls = chimpFlip("johnCalls", 0.9);
          else  johnCalls = chimpFlip("johnCalls", 0.05);

         /* Sample maryCalls according to alarm. */
        int maryCalls;
        if (alarm == 1) maryCalls = chimpFlip("maryCalls", 0.7);
           else maryCalls = chimpFlip("jaryCalls", 0.01);

        return maryCalls;
}
```

Translation Approach. Translating Chimple to PRAiSE is more involved than translating BLOG, since PRAiSE allows for arbitrary Java code. Our approach to the translation is to process the compiler representation of the run() method, and to modify the program so that instead of producing a sample it produces PRAiSE code. Then, the PRAiSE code is generated by executing the compiled method. In particular, we need to distinguish between conditions that involve random variables (possibly indirectly) and conditions that are independent of the random variables (e.g., tests of file content). The first type of conditions are translated into PRAiSE commands, whereas the second type are left untouched in the Java manipulation process and executed directly in the execution of the new Java method. Another special treatment is given to the addCost command in Chimple, which is translated into a corresponding factor in PRAiSE. Following is an example.

Chimple	PRAiSE code generator
	/* Code streamed to "out'*/ BufferedWriter out = ...;
int burglary = chimpFlip("burglary", 0.003);	out.write("random burglary: -> Boolean;"); out.write("burglary " + 0.003 + ";");
int earthquake = chimpFlip("earthquake", 0.002);	out.write("random earthquake: -> Boolean;"); out.write("earthquake " + 0.002 + ";");

int alarm;
if (burglary == 1) {
if (Earthquake == 1) alarm =
chimpFlip("alarm", 0.95);
else alarm = chimpFlip("alarm", 0.94);
} else {
if (earthquake == 1) alarm =
chimpFlip("alarm", 0.29);
else alarm = chimpFlip("alarm", 0.001);
}
	out.write("random alarm: ->
	Boolean;");
	out.write(
	"if (burglary) then
	if (earthquake) then alarm 0.95
	else alarm 0.94
	else
	if (earthquake) then alarm 0.29
	else alarm 0.001");

int johnCalls;
if (alarm == 1)
johnCalls = chimpFlip("johnCalls", 0.9);
else
johnCalls = chimpFlip("johnCalls", 0.05);
	out.write("random johnCalls: ->
	Boolean;");
	out.write("if (alarm) then johnCalls 0.9
	else johnCalls 0.05;");

int maryCalls;
if (alarm == 1) maryCalls =
chimpFlip("maryCalls", 0.7);
else maryCalls =
chimpFlip("maryCalls", 0.01);
	out.write(
	"random maryCalls: -> Boolean;");
	out.write("if (alarm) then maryCalls 0.7
	else maryCalls 0.001;");

if (maryCalls == 0) addCost(Double.POSITIVE_INFINITY);	out.write("if (not maryCalls) then praise_cost 0;");

The PRAiSE program resulting from the execution of the modified run() is the following:

```
random burglary: -> Boolean;
burglary 0.003;

random earthquake: -> Boolean;
earthquake 0.002;
random alarm: -> Boolean;
if (burglary) then
   if (earthquake) then alarm 0.95 else alarm 0.94
else
    if (earthquake) then alarm 0.29 else alarm 0.001;

random johnCalls: -> Boolean;
if (alarm) then johnCalls 0.9 else johnCalls 0.05;

random maryCalls: -> Boolean;
if (alarm) then maryCalls 0.7 else maryCalls 0.001;
if (not maryCalls) then praise_cost 0;
```

A nontrivial translation that we support is in the case of for loops. We provide two types of translations. “Lifted translation” is done when the loop defines variables in a symmetric way (that is, all the random variables are constructed from distributions with the same parameters). In this case, we replace the content of the for loop with a counter increment, so as to obtain the number of defined variables. The second kind of a loop is where our symmetry test fails, and then we apply the loop while replacing indexed variables (e.g., X[i]) with unique names (e.g., X_i).

Following is an example of a lifted translation.

Chimple	PRAiSE code generator

for (int i = 0; i < house_number; ++i) {
  String house_name = "Alarm" +
(new
  Integer(i)).toString();
  if (Burglary == 1) {
  if (Earthquake == 1) {
  Alarm[i] =
  chimpFlip(house_name,
bias3);
  } else {
  Alarm[i] =
  chimpFlip(house_name,
0.80);
  }
  } else {
  if (Earthquake == 1) {
  Alarm[i] =
  chimpFlip(house_name);
  } else {
  Alarm[i] =
  chimpFlip(house_name,
bias3);
  }
  }
}
	/* Count the domain size */
	int domSize = 0;
	for (int i = 0; i < house_number;
	++i) {
	  ++domSize;
	}
	
	/* Lifted translation */
	out.write("sort Domain_alarm: " +
	domSize
	  + ", alarm_2, alarm_1,
	alarm_0;\n");
	out.write("random alarm:
	Domain_alarm ->
	  Boolean;");
	out.write(
	  "if (burglary) then " +
	  "if (earthquake) then " +
	"alarm(X) " +
	  bias3 +
	  " else " + "alarm(X) 0.80" +
	  " else " +
	  "if (earthquake) then " +
	"alarm(X) 0.5"
	  + " else " + "alarm(X) " + bias3
	+ ";");

/* Observations */
if (Alarm[0] == 0)
  addCost(Double.POSITIVE_INFINITY);
if (Alarm[1] == 0)
  addCost(Double.POSITIVE_INFINITY);
if (Alarm[2] == 1)
  addCost(Double.POSITIVE_INFINITY);
	/* Observations */
	out.write(
	  "random praise_cost: ->
	Boolean;\n");
	out.write("praise_cost 1;\n");
	
	out.write(
	  "if (not alarm(alarm_0)) then " +
	"
	  praise_cost 0");
	out.write(
	  "if (not alarm(alarm_1)) then " + "
	  praise_cost 0");

	out.write("if (alarm(alarm_2)) then " + " praise_cost 0" + ");

Following is an example of a non-lifted translation.

Chimple	PRAiSE code generator

for (int i = 0; i < house_number; ++i)
{
String house_name = "Alarm" +
(new
Integer(i)).toString();
if (Burglary == 1) {
if (Earthquake == 1) {
Alarm[i] =
chimpFlip(house_name,
bias3);
} else {
Alarm[i] =
chimpFlip(house_name,
0.80);
}
} else {
if (Earthquake == 1) {
Alarm[i] =
chimpFlip(house_name);
} else {
Alarm[i] =
chimpFlip(house_name, i *
bias3);
}
}
}
	for (int i = 0; i < house_number; ++i)
	{
	out.write("random " + "alarm_" + i
	+ ": ->
	
	Boolean;");
	out.write(
	"if (burglary) then " +
	"if (earthquake) then " +
	"alarm_" + i +
	+ bias3 +
	" else " + "alarm_" + i + " 0.80"
	+
	" else " +
	"if (earthquake) then " +
	"alarm_" + i + " 0.5" +
	" else " + "alarm_" + i + " " + (i +
	1) *
	bias3 + ";");
	}

The implementation is based on a Java parser open sourced by Google.[3] Our translation is based on reading and manipulating the Abstract Syntax Tree (AST), exporting the new AST object into Java code, and finally compiling and executing by standard Java Virtual Machine (JVM) machinery.

[3] https://code.google.com/p/javaparser/

2.2 Phase 2: Probabilistic Programming in Datalog

Datalog is a well-known and well-studied query language over a database. This language is found appealing due to its combination of expressive power, simple syntax, and the fact that it is fully declarative: it features semantic independence from the algorithmic evaluation of rules, and semantic invariance under logical program transformations. While pursuing an IRL between probabilistic programs and solvers, we have initiated an exploration of extending Datalog with the concept and capabilities of probabilistic programming. The core question we first investigated is: what would be a natural extension of Datalog to support common probabilistic programming, and how can such an extension retain the robust declarative nature of Datalog? We pursue an extension that supports the common sense of probabilistic programming (and not just randomness in logic), and satisfies inherent features of Datalog. Specifically, we focus on three elementary conditions.

1. **Compositional probabilistic programming**. The extension should allow for random number generation, and should be able to use the random numbers as the parameters of subsequent numerical distributions.
2. **Recursion through randomness**. We should not limit recursion in Datalog (e.g., by requiring acyclic or stratified programs) to support probabilistic programming, as recursion is a central factor in the expressiveness and appeal of Datalog.
3. **Invariance under logical equivalence**. A logical program with randomness can be thought of as a First Order Logic (FO) program with function calls, and as such, the semantics should be invariant under logical equivalence (allowing, e.g., for rewriting programs for the sake of performance optimization).

Interestingly, it turns out that using existing approaches that incorporate randomness into logic violate at least one of the above three conditions. For example, languages that follow the "distributional semantics" such as ProbLog violate the first condition, logic based PPSs such as BLOG violate the second condition, and template-grounding language such as MLN and PRAiSE violate the third.

2.2.1 Theoretical Background

In a published paper [2], we presented such an extension of Datalog, namely *Probabilistic Programming Datalog*, or just *PPDL* for short. Our proposed extension provides convenient mechanisms to include common numerical probability functions; in particular, conclusions of rules may contain values drawn from such functions. When applying an ordinary Datalog program to a database, the result is a new database referred to as the *outcome*. When applying a PPDL program to a database, the result is a *probability distribution* over possible outcomes. In a nutshell, these possible outcomes are minimal solutions with respect to a related program (which we do not defined here) that involves existentially quantified variables in the conclusions (left hand sides) of rules. For more details, we refer the reader to our publication [2]. Observations are naturally incorporated by means of integrity constraints over the extensional and intentional relations.

We focus on programs that use discrete numerical distributions, but even then, the space of possible outcomes may be uncountable (as a solution can be infinite). We define a probability measure over possible outcomes by applying the known concept of *cylinder sets* [2] to a probabilistic chase procedure. We show that under our semantics, the meaning of a program is invariant under different chases and under rewritings that preserve logical equivalence. We also identify conditions guaranteeing that all possible outcomes are finite (and then the probability space is discrete).

2.2.2 The PlogiQL Language

We now describe how we extend LogiQL with PPDL. A key design principle is that both the probabilistic program instructions *and* the inference (over the random executions) should reside together, alongside LogiQL code. This is challenging, since in probabilistic programming, instructions are inherently different from inference actions, as the former are not required to execute (conceptually, at least) whereas the latter should execute just like ordinary (deterministic) code. This integration is achieved by introducing the lb:sample variable/object type. A probabilistic program is always associated with a value of type lb:sample, and currently, inference is done by producing a sample (e.g., MAP inference) or a set thereof. We first introduce our running example.

As a running example, we will use logistic regression. Our Extensional Database (EDB) predicates[4] contain the following.

```
entity(x) -> int(x).
feature(i) -> int(i).
ftr[x,i]=f -> entity(x), feature(i), float(f).
```

The goal is to predict the class of an entity, based on its features. We will use the *logistic function* for class prediction. A class of an entity x is either 0 or 1. The probability of 1 is given by:

$$\frac{1}{1+exp\left(\sum_i w_i \cdot ftr[x,i]\right)} \tag{3}$$

The coefficients w_i are not known in advance, and the goal of our program is to learn those by finding the best fit (most likely values) for a set of examples. The examples are given by the following EDB.

```
obs[x]=c -> entity(x), int(c), (c=0 ; c=1).  // Training examples
```

We assume (for some technical reasons that get clarified later) than we know some lower and upper bounds for the coefficients.

[4] EDB predicates are predicates in LogiQL that are not defined using rules, but rather given as external input to the database computations.

```
low[i]=a -> feature(i), float(a).  // Lower bound for parameter w_i
high[i]=a -> feature(i), float(a).  // Upper bound for parameter w_i
```

Type lb:sample. We define a special predicate type lb:sample. A predicate of type lb:sample has both a distinguished predicate type and a distinguished element type. We call those an lb:sample predicate and an lb:sample object.

In our example the lb:sample predicate is declared as follows.

```
smpl(s) -> .
lb:sample[`smpl].
```

A *PPDL* (or *random*) predicate is a predicate P that contains precisely one attribute from an lb:sample predicate. We allow only lb:sample and PPDL predicates to contain lb:sample objects. In our running example, we define the following PPDL predicates.

```
w[s,i]=w -> smpl(s), int(i), float(w).
sum[s,x]=w -> smpl(s), int(x), float(w).
prob[s, x] = p -> smpl(s), int(x), float(p).
class[s,x]=c -> smpl(s), int(x), int(c), (c=0 ; c=1).
```

Introducing Random Numbers (Prior). PPDL predicates can be generated using random number generators, using the syntax of ~Dist(params), where Dist is a parameterized distribution and params is a sequence of parameters. In our running example, we insert random values into our PPDL predicates as follows:

```
// Select a random coefficient per feature
w[s,i] = ~Uniform(low[i],high[i]) <- smpl(s), low(i), high(i).


// Select a random class per object
sum[s, x] = v <- agg<<v = total(z)>> z = w[s, i] * ftr[x, i].
prob[s, x] = v <- v = 1.0f / e, e = 1f + float:exp[sum[s, x]].
class[s,x]= ~Flip(p) <- sum[s,x]=_, p=prob[s, x].
```

Introducing Observations (Posterior). Observations serve as restrictions (constraints) on samples. A sample is valid if and only if it satisfies the observations. LogiQL will not make visible any sample that violates the constraints. Note, however, that the production of valid samples may required nontrivial solver capabilities. Observations have the syntax of constraints, except that we use the ~> notation rather than the -> notation. In our running example, we use observations to incorporate the training examples.

```
obs[x]=b, smpl(s) ~> class[s,x]=b.
```

Introducing samples. Samples can be generated by means of simple object construction. At least conceptually, every sample requires a solver to properly instantiate all of the PPDL predicates. In particular, introduction of a new sample may be an expensive operation. Using techniques like Markov Chain Monte Carlo (MCMC), this execution cost can be shared among multiple generated samples.

The actual values of the abstract sample type, (e.g., smpl) are accessed through operations on the abstract data type. The program is syntactically checked that concrete values of the sample type do not pollute the rest of the database -- the user can use the sample (subject to stratification), but cannot "export it" to the rest of the database without using the appropriate interface. This is how we introduce samples to our smpl predicate in our running example.

```
//To extract a value from samples, we use the lb:get_sample[`smpl]
// function that given an integer i, returns the i-th sample
my_class[i,x]=v <-
      int:range(0,10,1,ith),
      class[ lb:get_sample[`smpl][ith], x]  = v, entity(x).

hist_class[v]=c <- agg<<c=count()>>
      store(st), store_to_sample_number[st]=ith,
      class[ lb:get_sample[`smpl][ith],x ] = v.
```

Stratification. Consider an lb:sample predicate SP. We distinguish between stata that involve SP, as follows.

- The *core* stratum of SP is the stratum that includes all of the random number generations ~Dist(params) and the observations (~>) that include SP.
- A *high* stratum of SP is a stratum above the core stratum. Note that high stratums for SP do not involve any randomness or observations that relate that affect SP.

We use the following restriction in the core stratum: every lb:sample variable in the body of the rule must appear in the head. This restriction implies that the body of a rule can contain at most one lb:sample variable, since the head can include a single sample attribute. This restriction does not hold for high stratum of an lb:sample predicate.

Probabilities and MAP Inference. An important feature of the lb:sample type is that every lb:sample object *s* is associated with a probability, which is accessible via lb:prior(*s*). This probability is the **prior probability** of *s*; that is, this probability does not take into account normalization due to observations. Hence, lb:prior(*s*) is the product of the probabilities associated with the random values generated in the PPDL predicates that use *s*. For example, the following rules find the samples with the maximal probability among those generated in the core stratum.

```
likely(s) -> smpl(s).
maxProb[]=p -> float(p).
maxProb[]=p <- agg<<p = max(v)>> smpl(s), v=lb:prior(s).
```

```
likely(s) -> lb:prior(s)=maxProb[].
```

Another construct is a special second-order function lb:map (maximum a posteriori) is defined over lb:sample predicates. Applying lb:map(`SP), where SP is an lb-sample predicate, introduces and returns a new sample object of SP that is the most likely sample (or an approximation thereof), or more formally, an object of SP such that lb:prior(*s*) is maximal. For example, the following rules find a most-likely sample and its associated coefficients.

```
wGood[i]=w -> feature(i), float(w).
wGood[i]=w <- w[s,i]=w, s=lb:map(`smpl).
```

Comment: The functionality of lb:map(`SP) is different from that of selecting, among the sample objects of SP, the object *s* with the maximal lb:prior(*s*). This is true, since the existing samples need not necessarily contain a sample with a maximal probability. Moreover, the algorithm used for implementing lb:map(`SP) may be very different from that of generating samples and taking the one with the maximal probability.

We use the following restriction. For an lb:sample predicate SP, lb:prior and lb:map may be used only in high strata of SP. For example, the following program is illegal.

```
wGood[i]=w <- w[s,i]=w, s=lb:map(`smpl).
newW[s,i] = ~Normal(wGood[i],1]) <- smpl(s), feature(i).
```

Note that such a program would be legal if a different lb:sample predicate would be used, for example:

```
smpl2(s) -> .
lb:sample[`smpl2].
wGood[i]=w <- w[s,i]=w, s=lb:map(`smpl).
newW[s,i] = ~Normal(wGood[i],1]) <- smpl2(s), feature(i).
```

2.2.3 End-to-End Example

Based on the chunks we showed in the previous section, we now show an end-to-end example, including the translation into a rejection sampler in LogiQL (the LogicBlox query language) version 4.0.

```
//--- Type declarations ---//
// EDBs: mu, sigma, ftrs of entities x
low[i]=a -> int(i), float(a).
high[i]=a -> int(i), float(a).
ftr[x,i]=f -> int(x), int(i), float(f).
obs[x]=c -> int(x), int(c), (c=0 ; c=1).
// Probabilistically defined
sample(s) -> int(s).
lb:sample[`sample]   //--- Eliminated in Rewriting ---//
coeff[s,i]=w -> int(i), float(w), int(s), sample(s).
class[s,x]=c -> int(x), int(c), (c=0 ; c=1), int(s), sample(s).
sum[s,x]=w -> int(x), float(w), int(s), sample(s).
sum[s, x] = v <- agg<<v = total(z)>>  z = coeff[s, i] * ftr[x, i].

p[s, x] = v -> int(s), int(x), float(v), sample(s).

p[s, x] = v <- v = 1.0f / e,  e = 1f + float:exp[sum[s, x]].
```

//--- LogiQL/PPDL ---//	//-- Rejection-Sampling Rewriting --//
coeff[s,i] = [Uniform(low[i],high[i]) <- low(i), high(i), sample(s).	coeff[s, i] = a <- series<<a = rnd_uniform_real<l, h, seed>[s]>> l=low[i], h = high[i], seed = i, sample(s).
class[s,x]= [Flip(p)] <- sum[s,x]=_, p=1/(1+exp(sum[s,x])).	class[s,x] = c <- series<<c = rnd_binomial<1, p, seed>[s]>> p = p[s, x], seed = x, sample(s).
obs[x]=b, sample(s) ~> class[s,x]=b.	bad_sample(s) <- obs[x]=b, sample(s), class[s, x]!=b. good_sample(s) <- sample(s), !bad_sample(s).

Note that the top part of the program is common to both programs, except for the line lb:sample[`sample] that will be eliminated in the rejection rewriting. In the bottom part, the left side is the original PLogiQL program, and the right side is the rewriting, which goes line by line. Observe that we use the series<<>> function (predicate-to-predicate function, or “p2p” in our terminology) for generating random numbers.

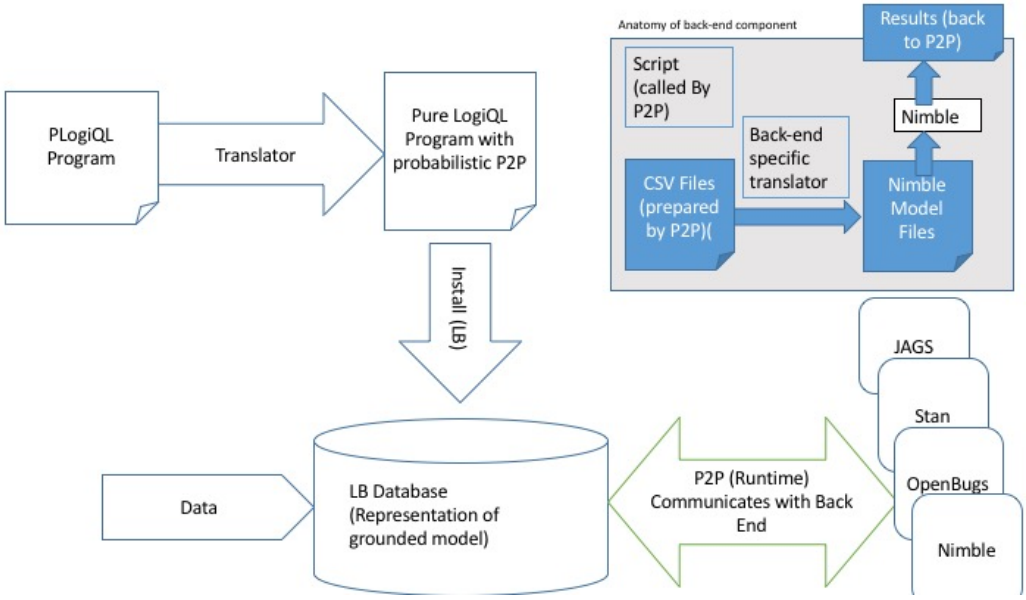


Figure 1: The PLogiQL System Design

2.2.4 Implementation

As illustrated in Figure 1, our current implementation of PLogiQL is based on a translation into LogiQL with calls for statistical (factor-graph) solvers for statistical inference. Specifically, we rewrite the given PLogiQL program into an ordinary LogiQL program that performs the following tasks:

1. Construct a factor graph that is represented as a LogiQL table.
2. Using a specialized p2p predicate, translate the table into the input of a factor-graph library. There are several libraries installed, and each comes with an adaptor for translating the corresponding table into input for the library.
3. Once the library is done with the inference (e.g., creating samples, computing marginal probability, etc.), the output of the library is translated back into PLogiQL predicates.

We currently use several libraries for statistical inference.

- **OpenBUGS.** A well-known, established, well accepted MCMC based solver that performs inference by automatically constructing a GIbbs sampler for the given model
- **Stan.** This library supports Hamiltonian Monte Carlo, Point based optimization (lbfgs) and variational inference. It produces a C++ program that performs inference. Our translation supports all the above. The main challenge was the fact that Stan's language is imperative which necessitated the translation of a declarative language to an imperative one.
- **Nimble.** This library supports extensibility by user defined distributions and/or inference methods. It also produces a C++ program that performs inference.

- **JAGS.** This is a more recent alternative to OpenBUGS that is supposedly easier to port in different platforms.

3 Results and Discussion

We now discuss the results of our research. Again, we consider each of the two phases separately.

3.1 Translating Languages to Solvers

Our experience in the translation of languages into solvers had been highly educating. We have invested a lot of effort in capturing fragments of the languages we considered (BLOG and Chimple) for the sake of translation. Each of the solvers focuses on some types of inference tasks, and it will take time before the solvers reach the generality of being able to support any full PPL.

Translation to PRAiSE. The main limitations of our current implementation are due to PRAiSE's limited support of *types*. In particular, the fact that PRAiSE supports only Boolean random variables allows us to translate programs with only Boolean and categorical random variables. In particular, the lack of numerical variable types in PRAiSE implies that the factors (probabilities) must all be fixed in the program, and not determined at runtime. Nevertheless, the PRAiSE team plans to extend their solution to support numerical (continuous or discrete) variables as part of the engagement with PPAML, and our team will extend the translation accordingly. Hence, we believe that the machinery we developed will be of significantly greater usefulness as it evolves into future versions alongside PRAiSE.

The current fragment of Chimple that we support for translation is very restricted. The restriction is due to two main reasons. The first is the same as for BLOG: PRAiSE's expressive power is currently limited to Boolean variable types. The second reason is that our translation has to have a full understanding of the program's flow in order to properly translate into PRAiSE. In particular, we support a limited syntax in "if" conditions.

Translation to BBVI. Our translation into BBVI has been less fruitful than the translation to PRAiSE. The main problem we have faced is that of accuracy, most likely due to the fact that our choice of a variational distribution has been suboptimal. Moreover, we have implemented the BBVI algorithm ourselves, and it is likely that more mature implementations from the BBVI team will lead to results of higher quality.

General sentiment. The first phase of our PPAML effort mainly educated us on different PPAML components. Yet, we have mainly concluded that translation into paradigm-specific solvers would entail a great engineering effort for every PPL, as sophisticated case analysis is required for understanding when each solver is relevant. This has led us to investigating a new language that matches our product philosophy, and which can lead to an intermediate representation.

3.2 PLogiQL

Our implementation of PlogiQL has progressed considerably within the PPAML program. It is not yet a complete end-to-end solution, since the links between the different components (Figure 1) are currently weak and make significant simplifying assumptions. Nevertheless, we are already able to solve a few problems. In particular, we are able to solve the following problems from PPAML's *small problem collection*:

- Problem 1: Bayesian Linear Regression
- Problem 2: Disease Diagnosis
- Problem 3: Hidden Markov Model
- Problem 8: Seismic 2D

In addition, we can solve Challenge Problem 2: Continent-Scale Bird Migration Modeling. Next, we show the results of a Naïve Bayes program.

3.2.1 Naïve Bayes for Retail Demand

We now summarize our experience with one of the experiments we are conducting with PLogiQL language and implementation.

Scenario. We are given daily sales data for a number of products (also known as *Stock Keeping Units*, or SKUs), in a number of stores (locations), together with some other data about sales, as well as some other indicators (e.g., what type of promotion was active for the given sku-store-day combination, what day of week (dow) or month of year (moy) did the sale take place on), and what sales class did the transaction belong to. See Table 1 for illustration. The sales class represents one of a number of possibilities of how many items are sold. Usually, a class represents one item sold, next class two items, and so on. Below we show a sample of data that the model is trained on. (Note in passing that we have obfuscated some of the data features as these come from date of a real retailer).

For a given dataset we summarize (in Figure 2) the corresponding graphical model. Gray nodes represent parameters we are ultimately trying to estimate, while yellow nodes represent random variables whose values are observed in the input data set containing the sales history.

Table 1: Example of retail input data

DOW_name	dow	sku_id	store_id	promo	moy	sales_class
Fri	1	1	1	1	1	1
Sun	5	10	2	2	5	3
Tue	3	10	2	2	4	2
Thu	2	2	2	1	1	2
Tue	3	2	2	1	2	3
Thu	2	2	2	1	2	3
Mon	4	2	2	1	2	3
Sun	5	2	2	1	3	3
Tue	3	2	2	1	2	3
Wed	6	2	2	1	2	3
Wed	6	2	2	1	3	3

Below, we show (with some details omitted) the PLogiQL program implementing the graphical model of Figure 2. For each row in the input, representing a day of sales at a given store, for a given product, we generate a sales class with a categorical distribution drawn from the predicate probSalesClass[w], which contains a vector of probabilities of a sale belonging to a sales class. Given a sales class c_i we use it to index into the set of class-indexed vectors of probabilities of generating Day of week (that is, the value probDowPerSalesClass[c_i]), to generate the day of week for the sale DOW_i from a categorical distribution (predicate generatedDow[c_i]).

```
//Declare a random world type for stochastic predicates
lang:randow(`World).
World(w) -> .


//vector 1..n of probabilities (where n is the number of sales classes)
//summing up to 1
probSalesClass[w] = p -> World(w), vector(p).
probSalesClass[w] ~ dirichlet(a) <- World(w), a = alphaClass[w].

//vector of 1.0 in the dimension of the number of sales classes
//to express the prior dirichlet(1, …., 1)
alphaClass[w] = v -> World(w), vector(v).
alphaClass[w] = v <-
      math << v = pack[s](i, z) >>
      s = num_salesClasses[],
      z = 1.0f,
      int:range(1, s, 1, i),
```

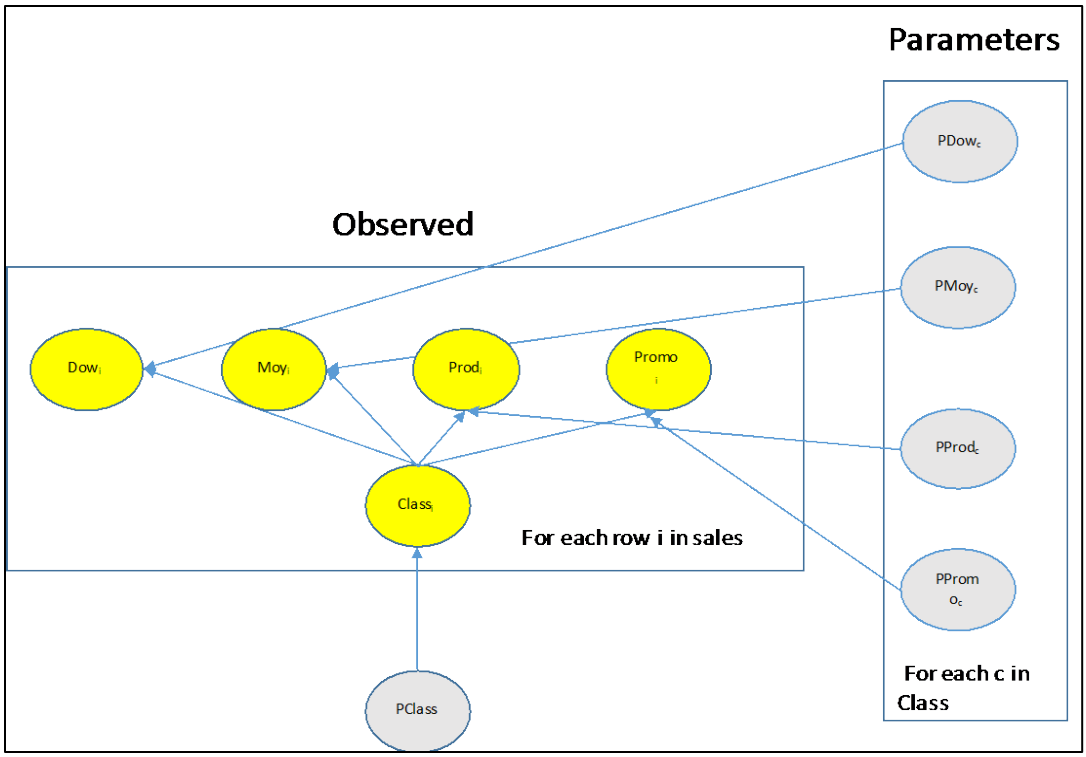


Figure 2: The graphical model for sales classification

```
        World(w).

//Vector of probabilities <1..M> (where m is the number of Days of Week
//expressing for each sales class the (conditional) probability of
//Day of week
probDowPerSalesClass[w, sc] = p -> World(w), salesClass(sc), vector(p).
probDowPerSalesClass[w, sc] ~ dirichlet(a) <-
        World(w), salesClass(sc), a = alphaDow[w].

alphaDow[w] = v -> World(w), vector(v).
alphaDow[w] = v <- //LogiQL primitive for creating a vector value
        math << v = pack[s](i, z) >>
        s = num_dows[],
        z = 1.0f,
        int:range(1, s, 1, i),
        World(w).

probSkuPerSalesClass[w, sc] = p -> World(w), salesClass(sc), vector(p).
probSkuPerSalesClass[w, sc] ~ dirichlet(a) <-
        World(w),
        salesClass(sc),
        a = alphaSku[w].

probStorePerSalesClass[w, sc] = p -> World(w), salesClass(sc), vector(p).
```

```
probStorePerSalesClass[w, sc] ~ dirichlet(a) <-
        World(w),
        salesClass(sc),
        a = alphaStore[w].

probPromoPerSalesClass[w, sc] = p -> World(w), salesClass(sc), vector(p).
probPromoPerSalesClass[w, sc] ~ dirichlet(a) <-
         World(w),
        salesClass(sc),
        a = alphaPromo[w].

probMoyPerSalesClass[w, sc] = p -> World(w), salesClass (sc), vector(p).
probMoyPerSalesClass[w, sc] ~ dirichlet(a) <-
        World(w),
        salesClass(sc),
        a = alphaMoy[w].


//For each sale datum (row) generate the sales class
generatedSalesClass[w, i] = v ->  World(w), int(i), salesClass(v)
generatedSalesClass[w, i] ~ categorical(p) <-
        World(w),
        probSalesClass[w] = p,
        int:range(1, num_sales[], 1, i).

//For each sale datum (row) generate the day of week, given the sales class
//for that row
generatedDow[w, i] = v -> int(i), dow(v), World(w).
generatedDow[w, i] ~ categorical(p) <-
        World(w),
        probDowPerSalesClass[w, sc] = p,
        sc = generatedSalesClass[w, i],
        s = num_sales[], int:range(1, s, 1, i).

generatedSku[w, i] = v -> int(i), sku(v), World(w).
generatedSku[w, i] ~ categorical(p) <-
        World(w),
        probSkuPerSalesClass[w, sc] = p,
        sc = generatedSalesClass[w, i],
        int:range(1, num_sales[], 1, i).

generatedStore[w, i] = v -> int(i), store(v), World(w).
generatedStore[w, i] ~ categorical(p) <-
        World(w),
        probStorePerSalesClass[w, sc] = p,
        sc = generatedSalesClass[w, i],
        int:range(1, num_sales[], 1, i).
```

```
generatedPromo[w, i] = v -> int(i), promo(v), World(w).
generatedPromo[w, i] ~ categorical(p) <-
      World(w),
      probPromoPerSalesClass[w, sc] = p,
      sc = generatedSalesClass[w, i],
      int:range(1, num_sales[], 1, i).

generatedMoy[w, i] = v -> int(i), moy(v), World(w).
generatedMoy[w, i] ~ categorical(p) <-
      World(w),
      probMoyPerSalesClass[w, sc] = p,
      sc = generatedSalesClass[w, i],
      int:range(1, num_sales[], 1, i).
```

Next, we connect the stochastic predicates such as generatedDow with supplied data. This is equivalent to conditioning the model with observations. In the current PLogiQL prototype implementation, we provide syntactic sugar by allowing the user to define a predicate named p_observation for any stochastic predicate p. This non-stochastic predicate's type signature must match the type signature of the corresponding stochastic predicate. Any value in the p_observation[p1,…,pn] will be used by the PLogiQL compiler to assign its value as the observation of the stochastic value in the predicate $p[w,p_1,\ldots,p_n]$. Missing values in p_observation simply mean that the corresponding stochastic value is unobserved.

```
generatedSalesClass_observation[i]=v -> int(i), salesClass(v).
generatedSalesClass_observation[i]=class <-
      actual_sales(i, _ ,_ ,_ ,_ ,class).
generatedDow_observation[i]=dow -> int(i), dow(dow).
generatedDow_observation[i]=dow <- sales_filter(i, dow,_ ,_ ,_ ,_ ,_).
```

Given the conditioning above, the system will automatically synthesize and perform inference tasks (in this case by invoking jags, OpenBugs, or Stan) and populate non-stochastic version of above predicates with results of the inference. For example, we may be interested in sampling from the posterior distribution (given the model and a particular set of sales observations).

```
probSalesClass_src[5][w,sc]=v -> World(w), salesClass(sc), float(v).
probSalesClass_src[w,sc]=v <- World(w),
      vector_index[probSalesClass[w], sc]=v,
      salesClass(sc).
      lang:sampled(`probSalesClass_src).
```

[5] We use the following convention: given a vector-valued stochastic predicate pred[w,…]=vec, we define the predicate pred_src[w,…,idx]=f, where the index idx is the value of the corresponding vector element in pred. This allow us to condition or pass around individual vector elements from a stochastic vector value in the rest of the program.

```
probDowPerSalesClass_src[w,sc,d]=v -> World(w), dow(d), salesClass(sc), float(v).
probDowPerSalesClass_src[w,sc,d]=v <- World(w),
        vector_index[probDowPerSalesClass[w,sc], d]=v, dow(d), salesClass(sc).
lang:sampled(`probDowPerSalesClass_src).
```

The definitions above notify the compiler that we are interested in samples from predicates probSalesClass_src and probDowPerSalesClass_src. These two predicates are simply lookups from the two vector-valued predicates probSalesClass and probDowPerSalesClass, which contain the probabilities of observing a sales class, and observing a particular day of week, given a sales class.

Again, as syntactic sugar, for every predicate marked as sampled, the system creates a pure LogiQL predicate containing the samples (or other results of inference), which we can query, analyze and treat as any other LogiQL (non-stochastic) predicate.

Table 2: Example contents of sampled predicates

SampleID	Class	Probability
1_1001	c1	0.0380493
1_1001	c2	0.11066
1_1001	c3	0.592816
1_1001	c4	0.197638
1_1001	c5	0.0608373
1_1002	c1	0.0343145
1_1002	c2	0.111968
1_1002	c3	0.584755
1_1002	c4	0.205151
1_1002	c5	0.0638114
1_1003	c1	0.034776
1_1003	c2	0.115098
1_1003	c3	0.57665
1_1003	c4	0.201499
1_1003	c5	0.0719761

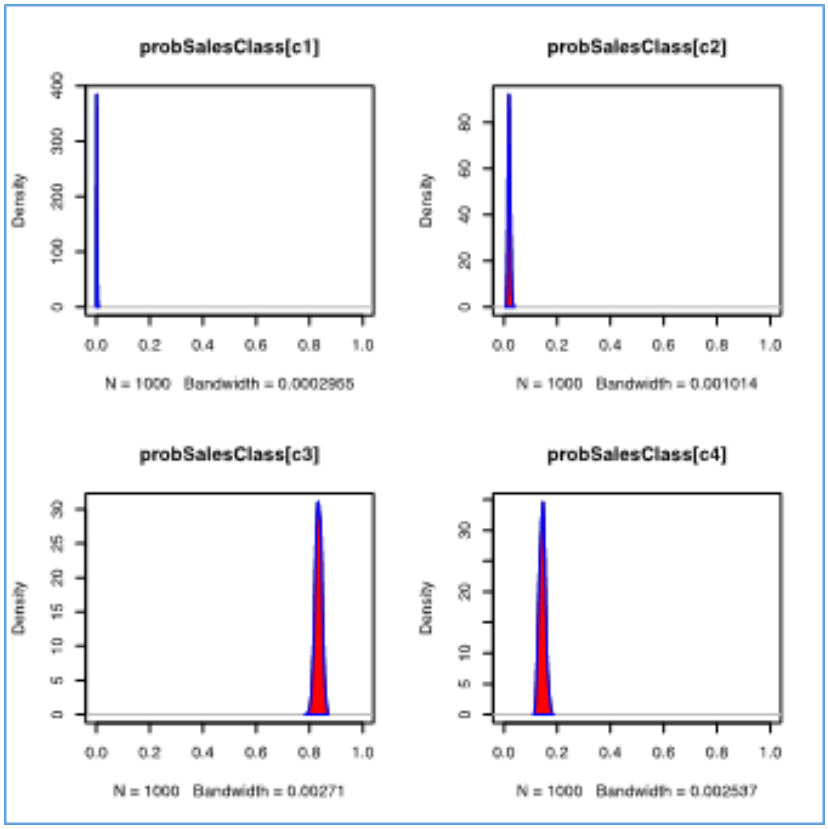


Figure 3: Sales class probability parameter probSalesClass (density from sample)

For example, for predicate probSalesClass_src, we can print out such a result predicate containing the results of MCMC sampling (for example, as in Figure 3).

The invocation of a back-end solver (in this case Stan) is managed automatically by the LogicBlox runtime. For example, having selected Stan and the back end solver, and sampling as the method of inference, when internal representation of the graphical model changes due to changes in input data (e.g., a new input sale row becomes available), the built-in LogicBlox mechanisms for incremental maintenance ensure that the appropriate Stan model is re-created, re-compiled, and that the sampler is re-run to update the values in the sample value predicates.

Next, we show some results of inference, on a set of 1000 sales records randomly chosen from a real retailer data set (data obfuscated to protect sensitive information). The goal of inference is to estimate various probabilities (of sale belonging to a particular class, of sales happening on a particular day of week, given a class, and so on). Once these parameters are estimated, they can be used to create a predictive model that can be used to forecast future sales.

In Figure 4, we plot the probability densities (estimated using kernel density estimator) for the values of the parameter probSalesClass in the thousand samples obtained from the Stan back end. The shape of the distribution of inferred values can be explained by the nature of the naïve Bayes model, as well as the fact that in the selected data set, we have very few sales in sales classes c1 and c2, the majority of the chosen sales belonging to classes c3 and c4. Similarly, for each sales class (c1 – c4) and each day of week, Figure

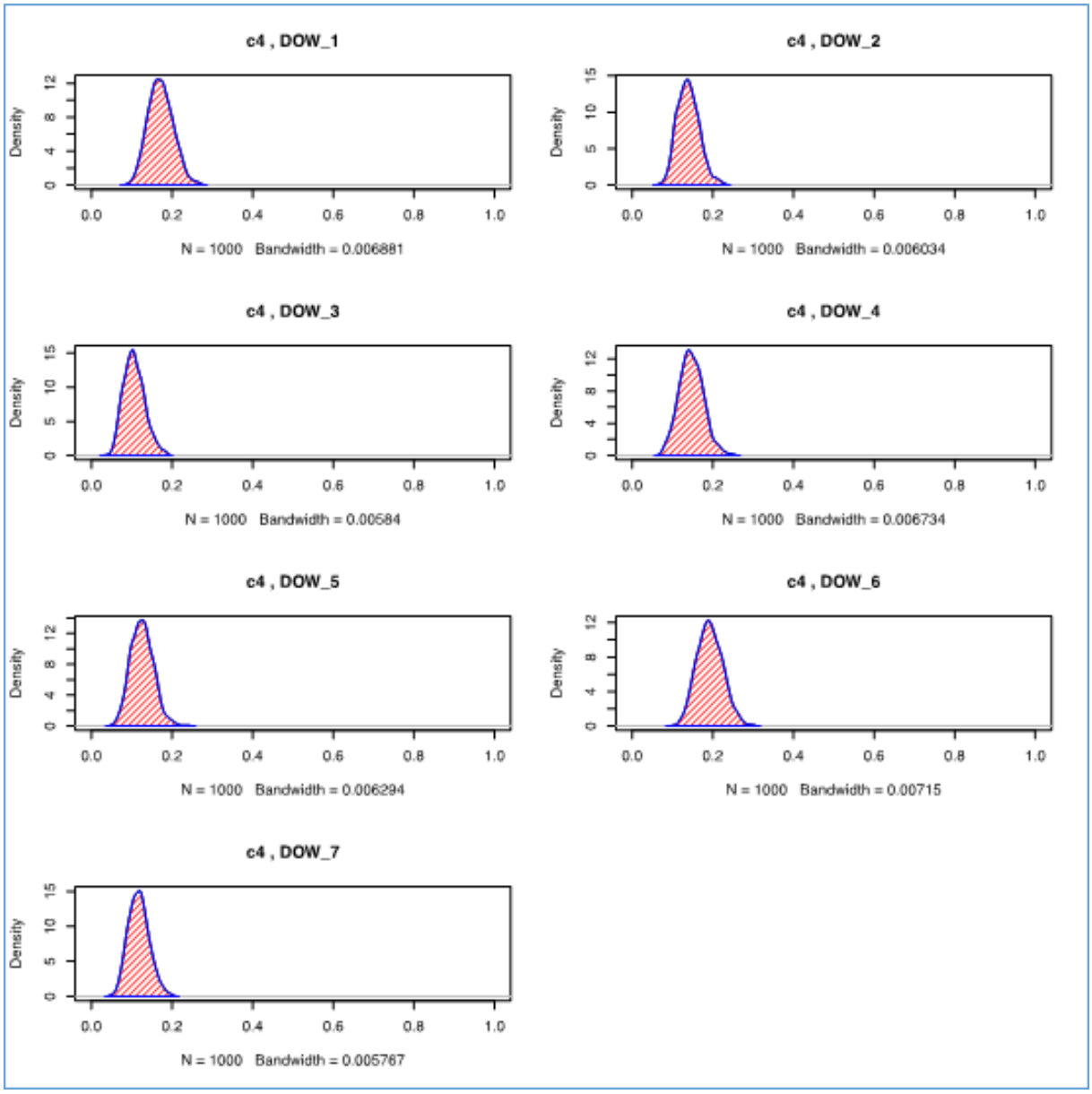


Figure 4: Estimation of the parameter probDowPerSalesClass giving the probability of a day of week, given a sales class.

5 shows the densities of the parameter which describes the probability of a given day of week being generated for a sales class.

If we examine the distribution (density) of the probability values across samples obtained, we notice that some of them (for example class c1, DOW_5 (Friday)) assume the distinct shape of the Gamma distribution (i.e., the shape of the prior). Examining the data set, we see that this is the result of having very little data in the original dataset, meaning that we cannot infer much about this variable (in the posterior).

By changing the parameters that let the system select which solver back end to use we may perform different kinds of inference. In the two sets of charts below, we show the probability of sales class using Stan optimization vs. Stan Hamiltonian base MCMC sampling. For example, with just one line of code, we can change the back end inference method (Stan sampling vs. Stan optimization to find modes). In Figure 6 we show, side

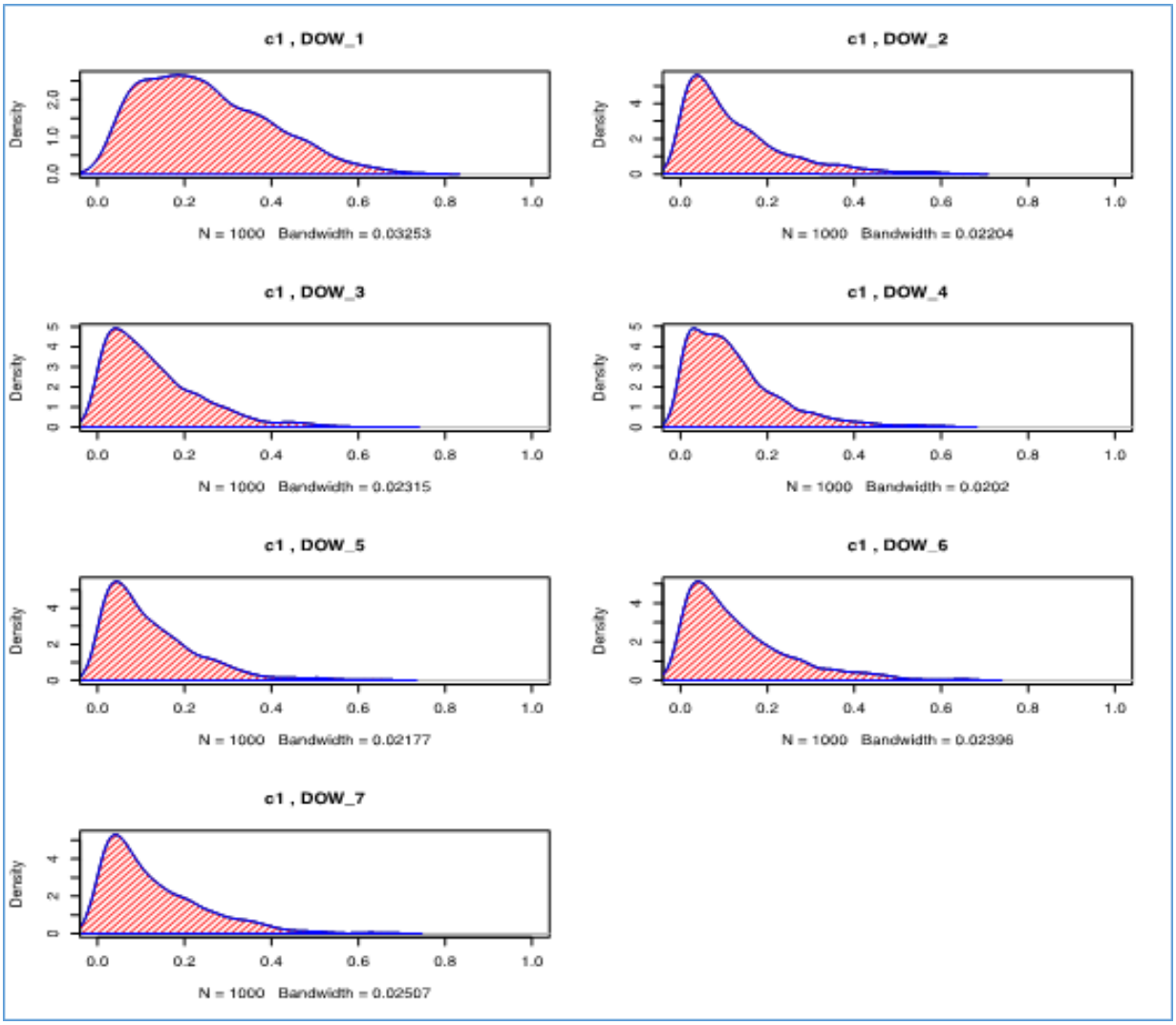


Figure 5: Probability of Day of Week for the class c1 (density from sample).

by side, the means of probabilities of belonging to a sales class (for the same data set) obtained with these two different inference methods. Similarly, in Figure 7 we show the optimization based and sample based estimated parameters for probDowPerSalesClass parameter of the model.

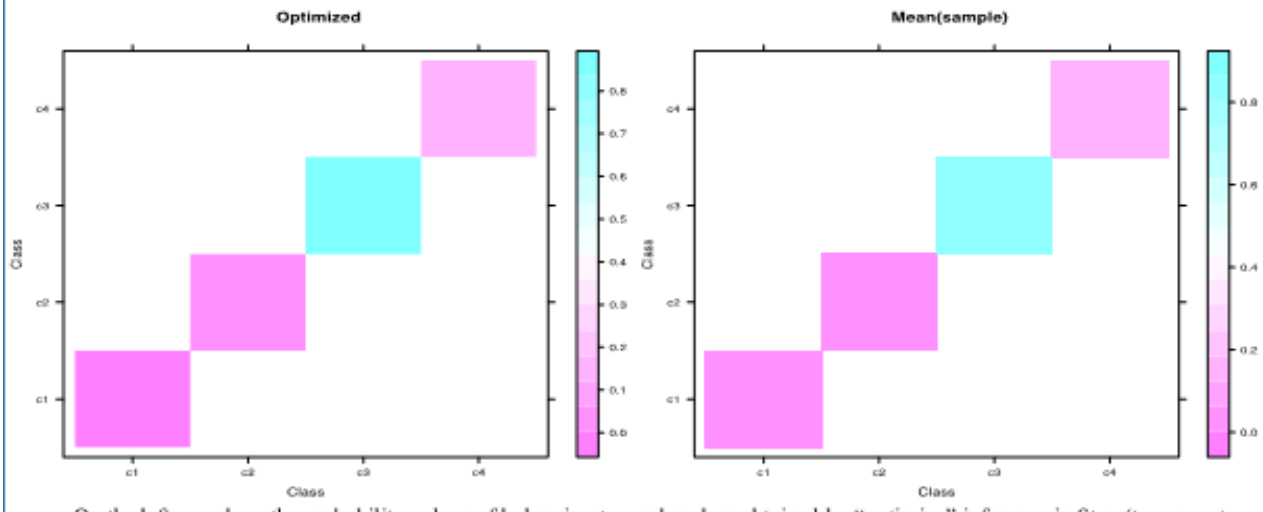
The above examples are just given to illustrate an intended workflow using probabilistic programming in PLogiQL. We are actively working on solving a number of challenges with our prototype implementation

1. We are working on issues of scaling the above (and similar) example to larger data sets, and on improving the performance of internal model construction (which is used to drive the back-end third party solvers such as Stan).
2. We are improving the language tools for translating PLogiQL into LogiQL (especially error messages and validity checking), and improving the engineering quality of the integration with the overall LogicBlox database management system.

4 Conclusions

The journey of our project within the PPAML program began with the task of translating PPAML programs (TA2) into PPAML solvers (TA3). We have conducted research, design, implementation and experimentation, and eventually concluded that an intermediate representation is a valuable effort to pursue. We have attempted to design an intermediate representation that is based on logic, where our in-house software can provide substantial heavy lifting. In doing so, we concluded that our platform can naturally incorporate probabilistic programming, and we have applied for a second phase of the PPAML program with the goal of developing a probabilistic-programming extension of Datalog. Such an extension required the development of the very basic foundations, an adaptation to the LogiQL language, and the engineering of a solution.

In the first phase, we focused on two PPLs, BLOG and Chimple, which are very different in nature. The former is logic based with a restricted syntax, while the latter is a Java package that can be used within arbitrary Java programs. Our translations focused on restricted fragments, as it would be impossible to achieve full language support in the scope of our engagement. We also worked with two solvers, PRAiSE and BBVI. Naturally, the translation from BLOG was easier (at least for the fragment we considered). Translating to PRAiSE imposed significant restrictions on the programs we could support since, at least at the time of our research, PRAiSE did not support numerical distributions. Nevertheless, PRAiSE was quite mature for categorical distributions. In the translation to BBVI, the main challenge was to find the variational distribution Q, and our choice seemed to result in results that lack sufficient accuracy. Overall, we have gained a substantial understanding of the challenges involved in the translations, as exposed in this document. We have also established clear directions for solvers to expand in order to facilitate the task of solving programs of PPLs, and we have shared these directions with the corresponding inventors.



On the left, we show the probability values of belonging to a sales class obtained by "optimize" inference in Stan (to compute modes of the posterior distribution). On the right are plotted the mean values of the samples obtained by Hamiltonian MCMC (same data set, same models) using Stan.

Figure 6: Two different inference method for the parameter probSalesClass

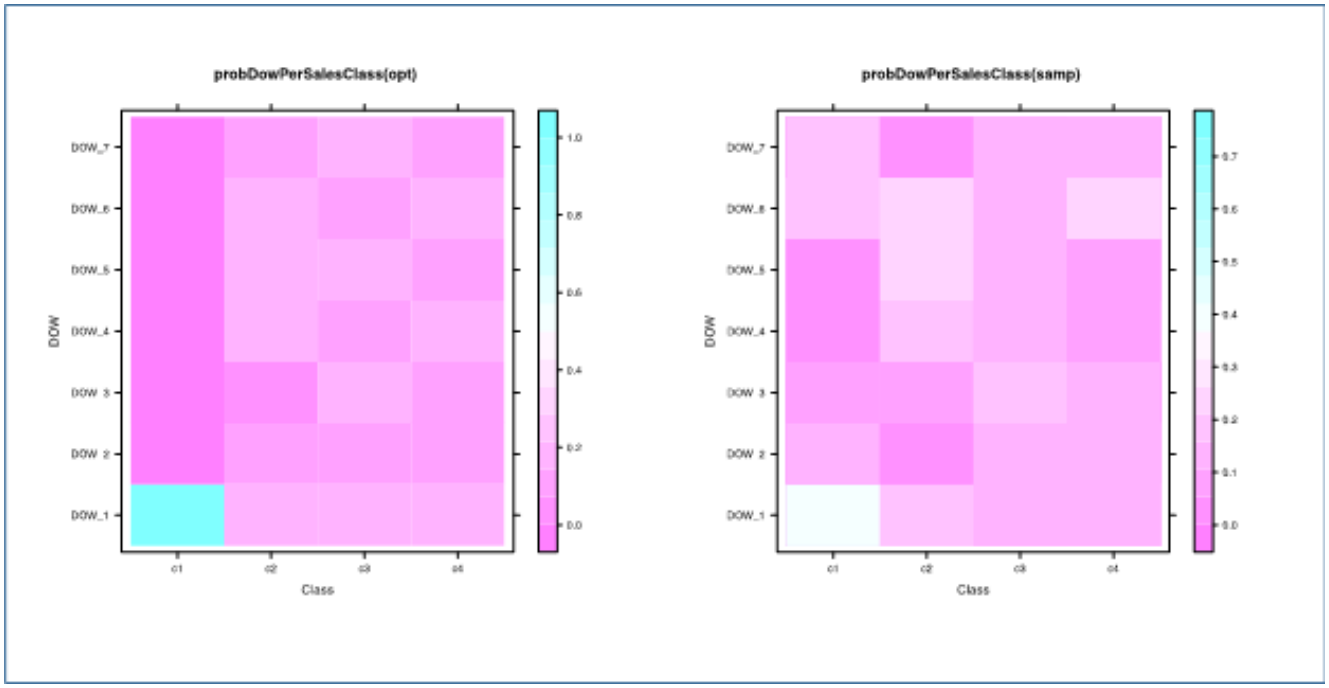


Figure 7: Estimated values (optimization vs. mean of samples) for probDowPerSalesClass parameter in the model.

Our development of probabilistic programming within Datalog involved several challenges. The first was the design of a language and semantics that allows for expressive probabilistic programming, recursion through randomness, and invariance under logical equivalence. Our development, PPDL, has been published in a conference paper, and an article version is currently under submission. The second challenge has been to incorporate PPDL in the PLogiQL language, in a way that allows for programming probabilistic models and making inference thereof in the same program. This has been achieved using the concept of the “randow” data type. The third challenge has been the engineering of an implementation. There are quite a few directions for continuation of this project, including the extension of the supported fragment of PPDL (beyond what is translatable to a factor graph), optimizing the choice of a solver based on the input, expediting the grounding process, and incorporating lifted inference.

5 References

[1] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382. ACM, 2015.

[2] R. B. Ash and C. Doleans-Dade. *Probability & Measure Theory*. Harcourt Academic Press, 2000.

[3] Vince Bárány, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, Zografoula Vagena: Declarative Probabilistic Programming with Datalog. ICDT 2016: 7:1-7:19

[4] Luc De Raedt, Angelika Kimmig: Probabilistic (logic) programming concepts. Machine Learning 100(1): 5-47 (2015)

[5] Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intel- ligence*. Synthesis Lectures on AI and Machine Learning. Morgan & Claypool Publishers, 2009.

[6] Terry Halpin and Spencer Rugaber. *LogiQL: A Query Language for Smart Databases*. CRC Press, 2014.

[7] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.

[8] Lyric Labs. Chimple. http://chimple.probprog.org/.

[9] B. Milch and et al. BLOG: Probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.

[10] R. Ranganath, S. Gerrish, and D. Blei. Black box variational inference. *Artificial Intelligence and Statistics*, 2014

[11] Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *IJCAI*, pages 1330–1339, 1997.

[12] SRI International. AIC-PRAiSE. http://aic-sri-international.github.io/aic-praise/.

6 LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
AST	Abstract Syntax Tree
BBVI	Black-Bock Variational Inference
DOW	Day Of Week
EDB	Extensional Database
ELBO	Evidence Lower BOund
FO	First Order Logic
ICDT	International Conference on Database Theory
IR	Intermediate Representation
IRL	Intermediate Representation Language
JVM	Java Virtual Machine
MCMC	Markov Chain Monte Carlo
MLN	Markov Logic Network
MOF	Month Of Year
PP	Probabilistic Programming
PPAML	Probabilistic Programming for Advancing Machine Learning
PPDL	Probabilistic Programming Datalog
PPL	Probabilistic Programming Language
PPS	Probabilistic Programming System
p2p	predicate-to-predicate
SKU	Stock Keeping Unit